Building a Knowledge Base of Rhythms

Charles Ames
info@charlesames.net
Orcid: 0000-0001-7263-3518
DOI: 10.46926/musmat.2022v6n2.72-94

Abstract: *This article is about composing programs. It explores generating up-front a knowledge base of rhythms as an alternative to generating rhythms on the fly. Since the knowledge base contains much more information than any one composing program will ever actually use, it employs ISAM technology to persist the information in one random-access file. The fundamental rhythmic entities are monophonic patterns and polyphonic textures. Patterns are defined as successions of pulse events (rests, attacks, ties). Textures present multiple patterns simultaneously. Various entity properties are calculated up front and stored alongside content data. For textures, these properties include an attacks profile, which is a vector. A persistent lookup map is realized to efficiently identify textures sharing attacks profiles in common. A pattern-to-pattern comparison identifies relationships which are documented in an ISAM map; these pattern-to-pattern relationships are then used to build a random-access map of texture-to-texture relationships. The article closes with a series of applications demonstrating how rhythms may be selected by combining knowledge-base queries with random shuffling and constraint filtering.*

Keywords*: Composing program. Rhythm knowledge base. ISAM.*

I. Introduction

Rhythm is the foundation of my music-theoretical thinking. To give a visual analogy, rhythm provides the shapes which pitch illuminates with color. Pretty much all of my composing programs lay out the rhythm first and fill in the pitches later. That applies to programs which emulate familiar styles and also to programs which explore pan-chromaticism and emancipated dissonance.

Most of my programs have taken an on-the-fly approach to generating rhythms. For example, my 1987 "Cybernetic Composer"[1] generated rhythms for rock, 2 jazz styles, and ragtime using context-sensitive grammars which divided long durations into shorter ones. Three of the four styles alternated 'composed' tunes with 'improvisatory' material; the 'composed' sections followed templates which generated 'fresh' rhythms for some segments and copied earlier rhythms for other segments. One feature of the rock style was an improvisatory break during which the accompaniment rested while the lead played through.

Received: October 21st, 2022
Approved: December 21st, 2022

[1]The "Cybernetic Composer" is described on my site at https://charlesames.net/cybernetic-composer/index.html.

You can judge from the examples provided on my site whether these grammars were effective; however, the site examples were selected to present my work in the best light. The two rock examples happen to present fairly active material for their improvisatory breaks, but this didn't always happen. The rhythm generator basically worked its way down the decision tree by flipping coins, and I was unable at the time to figure out how to incline those decisions toward more active results. Later it occurred to me that however many different rhythms my grammars were capable of producing, it would still be practical to run an enumeration algorithm that would list all candidates up front. Having that, my program could assign an activity score (attacks per beat) to each candidate. But by then the *Age of Intelligent Machines* exhibit was already on the road.

The idea of enumerating material up front stuck with me. I took a stab at it in the programs for my 1988 solo-violin piece, "Concurrence" [1]. Here the rhythmic material comprised different ways of dividing a quarter note into sixteenths, given that the instrument could do any of four things in each sixteenth: Initiate a new note, tie from a previous note, slur from a previous note, or rest. But this time around my concern wasn't for the qualities of specific patterns but rather for rather the degree of similarity between any two patterns. This was measured by calculating the minimum number of primitive operations required to transform one pattern into the other. The rhythm-selecting programs worked from a template which laid out the compositional form as a sequence of nodes, one per quarter note. These nodes were connected with references that indicated "this node is similar to that previous node" or "this node contrasts with that previous node".

The present effort generalizes what I did for "Concurrence" into a knowledge base of rhythms. First off, the idea of dividing quarter notes into sixteenths generalizes into the idea of dividing some unspecified longer duration into pulses of nominally-equal shorter durations. Beyond that, the fixed-length, monophonic patterns of "Concurrence" are here extended to textures with two, three, or four pulses, layering up to three patterns polyphonically. Hovering over this effort has been the prospect of combinatorial explosion. Faced with this prospect, the distinction between freshly attacked notes and slurred-to notes had to be dropped. I found it necessary to limit the maximum pattern length to 4 and to fix the number of layered patterns at 3 (fewer layers are accommodated by allowing some layers to rest). Even under these restrictions it required some 4 days to build the file; most of this time being taken up by the process of gleaning texture-to-texture relationships.

At this writing I have only the vaguest idea what kind of pieces this knowledge base will be used for, only that there will be more than one (health permitting) and that the first of these pieces will employ additive rhythms. Since the knowledge base is intended for multiple use by myself and possibly others (if there's any interest), its design is neutral and generic.

II. ISAM Persistence

According to my production framework[2] I should be implementing the overall compositional process as a succession of stages, with earlier stages leaving XML-formatted files of data for later stages to pick up. However here I am creating a knowledge base of building blocks, which knowledge base is potentially usable by many projects. Therefore the data needs to be persistent (stored in a file) and random-access. Although the knowledge base may hold a great deal of material, only a limited subset will be brought in to any individual application. Also there will be a need to query the knowledge base for sets of entities which meet certain criteria or for sets of entities which bear a certain relationship to some entity already in hand.

[2] https://charlesames.net/glossary/production-framework.html.

As a retired database programmer my first instinct was to go the full SQL route, but SQL databases impose considerable overhead. The requirements just listed can instead be satisfied using *Indexed Sequential Access Method* (ISAM), a technology central to SQL databases which is also available in stand-alone implementations. The ISAM implementation I found to do this in **Java** was **MapDB**.[3] **MapDB** is "free and open-source" and at this writing is into its 3.X production release. Despite this history the documentation remains sketchy. The online **Javadoc** equivalent simply lists methods without either method or parameter descriptions. Explanatory documents are few and far between, and of those that I could find, the sample code that did what I wanted to do wouldn't even compile.

That's the bad news. The good news is that **MapDB** implements a `BTreeMap` class, which acts like **Java**'s `SortedMap` interface but which stores its data in a random-access file rather than in memory. Like a `SortedMap`, a `BTreeMap` pairs *keys* with associated *values*. You can store **Java** objects as `BTreeMap` values with the proviso that the stored objects cannot reference other objects. So if object A relies on object B you have to implement object B with a unique identifier (**MapDB** seems to like long integers). Then if you've pulled object A from its `BTreeMap`, and want to look at a B contained within A, you can get object B's identifier from object A and use the identifier to pull object B from the BTreeMap where object B resides.

In the SQL world I would have stored each class of object in a *database table* with the unique identifier as its *primary key*. To facilitate queries for objects I would have implemented *table indices* enumerating the relevant properties and which objects have them.

MapDB has no tables and hence no table indices. However one can construct a `BTreeMap` which serves a purpose similar to a table index. In my first attempt to do this I built the *key* from one or more object properties and used the unique object identifier (a long) as the *value*. This didn't work because a `Map` (both `SortedMap` and `BTreeMap` implement this interface) associates exactly one value with each valid key. My second attempt incorporated the unique object identifier as the rightmost key element. This also didn't work — the sample code provided for this situation wouldn't compile — however I did manage to find a successful kludge by packing key elements into long integers.

III. Entities

The `PulseEventType` enumeration lists three things that one layer within a texture can do during a pulse: *`REST`*, *`ATTACK`*, and *`TIE`*.

Instances of the `PulsePattern` class persist in the knowledge base. A `PulsePattern` describes a succession of pulses and what happens in each pulse. This succession is implemented as an array of `PulseEventType` elements. The elements of this array are accessed using a `position` index, while the `length` of a `PulsePattern` instance is the array length (i.e. the number of pulses). Each `PulsePattern` is identified by a serially-generated long integer. There is only one constraint in forming a `PulsePattern`: a *`TIE`* can never follow a *`REST`*.

Instances of the `PulseTexture` class also persist in the knowledge base. A `PulseTexture` layers multiple simultaneous `PulsePattern` instances, all of the same `length`. This is implemented as an array of long integers, whose elements are `PulsePattern` identifiers. The elements of this array are accessed using a `layer` index, while the `depth` of a `PulseTexture` instance is the array length (i.e. the number of overlaid patterns). Each `PulseTexture` is identified by a serially-generated long integer. The definition of a `PulseTexture` abstracts away the order of patterns within the texture. Thus if a `PulseTexture` contains patterns A, B, and C, that would apply to pattern A on

[3] https://MapDB.org/.

top, pattern B in the middle, and pattern C on the bottom. However it would also apply to pattern B on top, pattern C in the middle, and pattern A on the bottom.

A specific *statement* of a `PulseTexture` is obtained by combining the `PulseTexture` with a mapping from musical parts to texture layers. For any given `PulseTexture` of `depth` 3, up to six distinct statements are available, corresponding to the six permutations of the set $\{0, 1, 2\}$. Part-to-layer mappings are represented in the knowledge base using the `Permutation` class. `Permutation` instances combine an array of 3 ordinals with a long-integer identifier derived by packing the number of array elements (3) in the leftmost nibble and the individual ordinals in successive nibbles thereafter. The six `Permutation` instances persist in the knowledge base, though in practice they are cached into an in-memory map.

Both simple and compound texture statements can be represented using the `TextureStatement` class, whose components are pairings of `PulseTexture` instances with `Permutation` instances. The `TextureStatement` class effectively presents a two-dimensional array of `PulseEventType` elements, with a `part` index ranging from 0 to `depth`-1 and a `position` index ranging from 0 to `length`-1. Instances of the `TextureStatement` class do *not* persist in the knowledge base; however the end product of any session working with the knowledge base will typically be one or more `TextureStatement` instances.

Instances of the `Relation` class persist in the knowledge base because in order to get **MapDB** to work I needed to associate each `Relation` instance with a persistent long-integer identifier. A `Relation` instance combines a `RelationCategory` with an integer `offset`. `RelationCategory` is a software enumeration which among other things includes code to determine whether two patterns are related in that way. The `offset` depends upon the `RelationCategory`. For examples the `offset` for `RelationCategory.`*`ROTATE`* indicates how far to right shift, while the `offset` for `RelationCategory.`*`MASK`* indicates which pulse position is affected. The full `RelationCategory` enumeration is presented below.

The remaining class whose instances persist in the knowledge base is `Profile`. A `Profile` is array of integers which count how often a certain `PulseEventType` (or combination thereof) occurs simultaneously during the corresponding pulse in a `PulseTexture` instance. The elements of this array are accessed using a `position` index, while the `length` of a `Profile` instance is the array length (i.e. the number of pulses). `Profile` instances also include a long-integer identifier which packs the number of array elements (the `length`) in the leftmost nibble and the individual counts in successive nibbles thereafter.

IV. Scalar Properties

The most fundamental property of a `PulseTexture` is `length`. The most fundamental properties of a `PulseTexture` are `length` and `depth`.

There is a family of scalar properties which have absolute versions, which are integer counts, and relative versions. The relative version is a floating-point number calculated as the count divided by its upper limit. Examples for the following definitions will be drawn from `PulseTexture` #16520, which has the following content:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶▶ – –]
2	78	[– – ▶]

The content is represented pictographically: "▶" means *ATTACK*, "−" means *TIE*, and " " means *REST*.

i. Attacks

The `attacksCount` and `attacksRatio` are dual properties shared by `PulsePattern` and `PulseTexture` instances. For `PulsePattern` the `attacksCount` is the number of pulses with the *ATTACK* event type, for which the upper limit is the pattern length. For `PulseTexture` the `attacksCount` remains the number of *ATTACK* pulses, however the upper limit expands to `length` × `depth`.

Referring back to `PulseTexture` #16520, `PulsePattern` #40 has 2 attacks, `PulsePattern` #54 has 2 attacks, and `PulsePattern` #78 has 1 attack. Therefore `PulsePattern` #16520 has an `attacksCount` of $2 + 2 + 1 = 5$ attacks. The `attacksRatio` is $5/12 = 42\%$ of a possible 12.

ii. Coverage

The `coverageCount` and `coverageRatio` are also dual properties shared `PulsePattern` and `PulseTexture`. For a `PulsePattern` instance the `coverageCount` is the number of *ATTACK* pulses plus the number of *TIE* pulses, for which the upper limit is the pattern length. For `PulseTexture` the upper limit expands to `length` × `depth`.

Referring back to `PulseTexture` #16520, `PulsePattern` #40 has 3 non-rest events, `PulsePattern` #54 has 4 non-rest events, and `PulsePattern` #78 has 3 nonrests. Therefore `PulsePattern` #16520 has a `coverageCount` of $3 + 4 + 3 = 10$. The `coverageRatio` is $10/12 = 83\%$ of a possible 12.

The compliment to `coverageCount` is the count of rests, which for `PulsePattern` #16520 is 2 or 17% of a possible 12.

iii. Dispersion

The `dispersionCount` and `dispersionRatio` are dual properties of `PulseTexture` instances. The `dispersionCount` is the number of pulses with an *ATTACK* in at least one layer. The upper limit is `length`.

`PulseTexture` #16520 has 1 attack in pulse-position 0 (`PulsePattern` #54), 2 attacks in pulse-position 1 (`PulsePattern` #40 and #54), 1 attacks in pulse-position 2 (`PulsePattern` #78), and 1 attack in position 3 (`PulsePattern` #40). This gives a `dispersionCount` of 4 with a 100% `dispersionRatio` out of a possible 4.

The compliment to `dispersionCount` is the count of pulses without an *ATTACK* in any layer. For `PulsePattern` #16520 this is 0 or 0% of a possible 4.

iv. Imbalance

The `imbalanceCount` and `imbalanceRatio` are dual properties of `PulseTexture` instances. The `imbalanceCount` is calculated by iterating through the layers, determining the maximum and minimum number of attacks, then subtracting the minimum from the maximum. If this max-min is zero, all layers will share the same number of attacks. Otherwise at least one layer will have more than its fair share activity. The upper limit obtains when all attacks happen in the same layer.

For `PulseTexture` #16520 the maximum number of attacks is 2 (`PulsePattern` #40 and #54) and the minimum number of attacks is 1 (`PulsePattern` #78). Thus the `imbalanceCount` is $2 - 1 = 1$ and the `imbalanceRatio` is 25% of a possible 4. An `imbalanceCount` of 4 (100%) would have resulted if one pattern had 4 attacks and another of the remaining 2 patterns had no attacks.

The compliment to `imbalanceCount` is `length` − `imbalanceCount`. For `PulsePattern` #16520 this is $4 - 0 = 4$ or 100% of a possible 4.

V. Order of Patterns in Textures

The enumeration algorithm for `PulseTexture` instances iterates through all possible pattern IDs for layer 0. Layer-1 pattern IDs range from the layer-0 pattern ID upwards, while layer-2 pattern IDs range from the layer-1 pattern ID upwards. This allows the same pattern to appear twice or three times in the same texture, but prevents any two textures from presenting the same set of patterns in different permutations.

Anyone employing a `PulseTexture` instance as a building block for a musical passage will quickly need to determine which musical part will play which layer. Accepting the default order is generally not a desirable option. If the voices are co-equal then a better option would be to permute layers randomly. If the context is metric and one voice has a lead role with the others providing accompaniment, then it may be desirable to assign those layers which most coincide with strong beats to the accompaniment and give the more syncopated layer to the lead voice. This metric option requires strong-beat position data.

Yet another option is to find the permutation which orders the patterns from most to least active. To quantify the level of activity within a pattern I implemented a *beauty contest*[4] using the formula:

$$\text{pattern activity} = 16 \times \text{attacks} + \text{ties} + \text{rand}[0, 1]. \tag{1}$$

For example consider `PulseTexture` #16520:

- [▶ − ▶] contains 2 attacks and 1 tie. A random offset of 0.524 gives an activity score of 33.524.
- [▶ ▶ − −] contains 2 attacks and 2 ties. A random offset of 0.340 gives an activity score of 34.340.
- [− − ▶] contains 1 attacks and 2 ties. A random offset of 0.721 gives an activity score of 18.721.

The permutation which orders the patterns from most to least active is therefore $(1, 0, 2)$. Notice that the number of attacks always swamps the number of ties, while the random offset exerts influence only when two patterns share the same counts of attacks and ties.

VI. Profiles

I adhere to the premise that musical meter is established through the convergence of polyphonic attacks on strong beats and divergence of attacks on weak beats.[5] The rhythmic knowledge base described here does not itself favor 'metric' textures; however, it does provide a handle which can be used to identify them. This handle is the *attack profile*.

[4] Beauty contests are simpler among the merit-based problem-solving strategies discussed in [2]. The specific heading is "'Beauty Contest' and 'Blackboard' Models", pp. 75–76.

[5] This premise underlies an article by Karl Kohn [3]. Kohn showed me a renotation example during an undergraduate composition lesson; however the lesson slipped from memory until we met again a few years ago. Convergence of polyphonic attacks is also the premise underlying my "Complementary Rhythm Generator". The backstory given on my site (`https://charlesames.net/rhythm/index.html`) says that I came to the premise while studying the collected motets of Guillaume de Machaut. I now realize that Kohn's lesson primed me for this realization.

The *attacks profile* is a vector property of the `PulseTexture` class. It is an array of small integers (actually bytes) giving the number of simultaneous `ATTACK` events, by pulse `position`. When two textures have the same attack profile, they will be heard to have the same aggregate rhythm. If the profile divides into regular groups initiated by larger `ATTACK` counts, then this aggregate rhythm will be heard as meter.

Two textures having the same attacks profile is a binary relationship which establishes *equivalence classes* among `PulseTexture` entities. Since `PulseTexture` entities have short lengths it is possible to pack their profiles into the 16 nibbles of a long integer, then create a lookup map of `PulseTexture` instances using this packed profile as the most significant key element.

For example, `PulseTexture` #16520 has attack profile $[1,2,1,1]$. A query of the lookup map returns 756 `PulseTexture` instances sharing the same attack profile. These results include `PulseTexture` #16520. Here is a random sampling of other textures returned by the same query:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶ ▶ – –]
2	78	[– – ▶]

`PulseTexture` #15176

Layer	PatternID	Content
0	39	[▶ –]
1	44	[▶ ▶]
2	69	[– ▶ ▶]

`PulseTexture` #11757

Layer	PatternID	Content
0	36	[▶▶]
1	40	[▶ – ▶]
2	60	[▶ – –]

`PulseTexture` #17033

Layer	PatternID	Content
0	41	[▶ – –]
1	42	[▶]
2	71	[– ▶▶▶]

`PulseTexture` #15097

Layer	PatternID	Content
0	39	[▶ –]
1	42	[▶]
2	71	[– ▶▶▶]

`PulseTexture` #17112

Layer	PatternID	Content
0	41	[▶ – –]
1	42	[▶ ▶]
2	71	[– ▶ ▶]

Yet to be implemented as of this writing are the *rest profile*, the *coverage profile*, and the *non-attacks profile*. The *rest profile* gives the number of simultaneous `REST` events by pulse. The *coverage profile* gives the number of simultaneous non-`REST` events (`ATTACK` or `TIE`) by pulse; it is the compliment of the *rest profile*. The *non-attacks profile* gives the number of simultaneous non-`ATTACK` events (`REST` or `TIE`) by pulse; it is the compliment of the *attacks profile*. Among these the *coverage profile* is most clearly useful.

VII. Relations

My original plan was to enumerate all possible `PulseTexture` instances of `depth` 3 and lengths of 3, 4, and 5, then evaluate all `PulseTexture` pairs to discover whether some close relationship existed between the pair and, if so, to document that relationship. This plan was wrecked by combinatorial explosion. Testing for `depth` 3 produced 120 `PulseTexture` instances of `length` 2 and 1,771 `PulseTexture` instances of length 3. This meant evaluating $(120 + 1{,}771)^2 = 18{,}912 = 3{,}575{,}881$ `PulseTexture` pairs. The test ran through to completion, but it took my laptop 3 days

to run it. As things turned out I was able to speed things up substantially by caching pattern maps into in-memory collections. This allowed me to include textures of `length` 4, of which there were 29,260 instances. The number of required pairwise evaluations thus increased to $(120 + 1{,}771 + 29{,}260)^2 = 311{,}512 = 970{,}384{,}801$; however in-memory caching allowed my laptop to crunch these through in 2 days rather than 3. However, there were bugs. Once those were (mostly) remedied it required over 4 days to build the file.

Relations are defined between `PulsePattern` instances. A `Relation` instance assigns a unique long-integer ID, to the combination of a `RelationCategory` (defined through an `Enum`) and an offset. Thus *IDENTITY*(0) indicates the identity relation while *REVERSE*(0) indicates the retrograde relation. 0 is the only offset permitted for these two categories. Here is a summary of `RelationCategory` items:

- *IDENTITY* — Target same as source. This relation only happens when the compared `PulsePattern` instances have the same ID.
- *REVERSE* — Target retrograde of source. This relation preserves durations, for examples, *REVERSE*(0) for [▶ − −] gives [▶ − −] while *REVERSE*(0) for [▶ − ▶] gives [▶ − ▶].
- *EXTEND* — Target same as source except for pulse inserted in N-th position. If an inserted *REST* would precede a *TIE*, then the *TIE* converts to an *ATTACK*. For example, *EXTEND*(1) for [▶ − ▶] gives [▶ ▶ − ▶], [▶ − − ▶], and [▶ ▶ ▶].
- *TRUNCATE* — Target same as source except for pulse removed from N-th position. For example *TRUNCATE*(1) for [▶ − − ▶] gives [▶ − ▶]. If the *ATTACK* is truncated from the succession *REST*, *ATTACK*, *TIE* then the *TIE* converts to an *ATTACK*. For example, *TRUNCATE*(2) for [▶ ▶ −] gives [▶ ▶].
- *ROTATE* — Target derived from source by right shifting N positions, with $N \neq 0$. For example *ROTATE*(1) for [▶ ▶ ▶] gives [▶ ▶ ▶]. If right-shifting a *REST* places it in front of a *TIE*, then the *TIE* converts to an *ATTACK*; for example *ROTATE*(1) for [− ▶] gives [▶ ▶] rather than the invalid [▶ −].
- *MASK* — Target derived from source by resting in N-th pulse. For example, *MASK*(1) for [▶ ▶ ▶ ▶] gives [▶ ▶ ▶]. If pulse $N + 1$ has a *TIE*, then the *TIE* converts to an *ATTACK*; for example, *MASK*(0) for [▶ − −] gives [▶ −]rather than the invalid [− −].
- *EXCHANGE* — Target derived from source by swapping pulse position N with pulse position $N + 1$. For example, *EXCHANGE*(1) for [▶ ▶ ▶] gives [▶ ▶ ▶]. Exchanges preserve durations; for example *EXCHANGE*(0) for [▶ −] gives [▶ −]. If the exchange would move a *REST* in front of a *TIE*, then the *TIE* converts to an *ATTACK*; for example *EXCHANGE*(0) for [−] gives [▶] rather than the invalid [−].

There is no exclusivity to relations: [▶ ▶ ▶ ▶]bears the *IDENTITY*(0) relation to itself, but it also bears the relations *REVERSE*(0), *ROTATE*(N) (for every N), *EXCHANGE*(N) (for every N) and so forth.

Coding the procedures which discover valid relations between patterns was a challenging exercise combining asymmetric iterations with conditional branching. The special circumstances introduced by ties greatly complicate things. This is an exercise I would recommend for beginning programmers, especially those who wish to pursue composing programs.

To document all pattern-to-pattern relations I implemented a `BTreeMap` whose *key* consisted of three long integers: the source-pattern ID, the relation ID, and the target-pattern ID. Placing the relation ID in the middle allowed map queries of the form: given a a reference instance, which `PulsePattern` instances bear any sort of relation? Also map queries of the form: given a a reference instance, which `PulsePattern` instances bear a specific relation? (The `BTreeMap` value repeated the relation ID.) For the record, the number of pattern-to-pattern relations discovered was 6889.

Two `PulseTexture` instances are defined to bear a `Relation` if all their layered `PulsePattern` instances bear the same relation. Given any two `PulseTexture` instances (a source and a target), the comparison algorithm iterated through all the different ways the target layers could align with the source layers. (With the knowledge base `depth` set to three layers per texture, this amounted to six permutations.) The algorithm then identified all the different relations existing between source layer 0 and its corresponding target layer. For each layer-0 relation, the remaining layers were compared. If both remaining source layers bore the same relation to their corresponding target layers, then the two textures were determined to have that relation.

To document the close relationships discovered by these pairwise comparisons, I created a `BTreeMap` whose *key* combined four long integers: the source-texture ID, the relation ID, the permutation ID, and the target-texture ID. (The `BTreeMap` value repeated the relation ID and the permutation ID.)

Specifying the source-texture ID as 16520 and allowing the remaining key fields to range freely queries all texture-to-texture relations with `PulseTexture` #16520 as the source. This query fetched back 51 instances in all. Here once again is `PulseTexture` #16520:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶ ▶ – –]
2	78	[– – ▶]

And here is a random sampling of 6 from the 51 instances fetched:

MASK(2)

`PulseTexture` #9688

Layer	PatternID	Content
0	34	[▶]
1	48	[▶ ▶ ▶]
2	76	[– –]

MASK(1)

`PulseTexture` #7003

Layer	PatternID	Content
0	32	[▶ ▶]
1	46	[▶ ▶ –]
2	65	[– ▶]

ROTATE(2)

`PulseTexture` #21902

Layer	PatternID	Content
0	69	[– ▶ ▶]
1	79	[– – ▶ ▶]
2	46	[▶ ▶ –]

ROTATE(3)

`PulseTexture` #29016

Layer	PatternID	Content
0	78	[– – ▶]
1	61	[▶ – – ▶]
2	69	[– ▶ ▶]

NOT(0)

`PulseTexture` #1958

Layer	PatternID	Content
0	42	[▶]
1	29	[ ]
2	30	[▶]

TRUNCATE(0)

`PulseTexture` #1684

Layer	PatternID	Content
0	19	[▶ – ▶]
1	20	[▶ – –]
2	23	[– ▶]

(Notice that `PulseTexture` #9688 is missing an attack in position 3 of layer 0. The `RelationCategory` code still needs work.)

VIII. Application: Foundational Textures

The rhythmic knowledge base described here takes a neutral attitude toward rhythmic material. With applications this changes. Here the user actively expresses some sort of preference. He or she first uses a `BTreeMap` to *query* for candidates meeting some desired criterion. The resulting list can then be *filtered* by iterating through them and discarding those which fail to meet additional criteria. The pared-down list should then be shuffled randomly to eliminate enumeration biases. If all criteria have been applied, then the first candidate in the shuffled list becomes the selection. However sometimes additional criteria remain which impose additional overhead — like trying out different permutations. In this scenario a second iteration may be necessary. The selected candidate will be the first one in the shuffled list which has a permutation that works.

i. Silence

A texture is *concerted* when for any given pulse position, all layers have the same event type. This very first application will demonstrate how to identify concerted textures which are entirely silent; that is, where the event type is *`REST`* for all layers and all positions.

There is no lookup map which specifically identifies silent textures. The best route available is the attacks-profile lookup map. Querying this map for the attacks profile [0,0] returns 10 candidates (from the 120 textures of length 2). The first 5 of these are:

`PulseTexture` #0

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	0	[ ]

`PulseTexture` #7

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	7	[– –]

`PulseTexture` #5

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	5	[–]

`PulseTexture` #30

Layer	PatternID	Content
0	0	[ ]
1	5	[–]
2	5	[–]

`PulseTexture` #30

Layer	PatternID	Content
0	0	[ ]
1	5	[–]
2	7	[– –]

While it happens that `PulseTexture` #0 is the exactly the texture sought, the end user can't be expected to know that the texture-enumeration algorithm would have produced this first. However, with only 10 candidates it is not unreasonable to iterate through the candidates to filter out those with event type *`TIE`* in any layer or position. Once coverage profiles have been captured within the stored `PulseTexture` instances, it will be a simple matter to exclude textures with coverage profiles other than [0,0].

To identify silent textures of length 3 involves first querying for textures with attacks profile [0,0,0]. This returns 20 candidates (from the 1771 textures of length 3), which can then be filtered for coverage profile [0,0,0].

Likewise, identifying silent textures of length 4 involves first querying for textures with attacks profile [0,0,0,0]. This returns 35 candidates (from the 29260 textures of length 4), which can then be filtered for coverage profile [0,0,0,0].

In summary, the query-and-filter operations just described produce exactly three examples of entirely silent textures, one for each texture length in the knowledge base:

`PulseTexture` #0

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	0	[ ]

`PulseTexture` #120

Layer	PatternID	Content
0	8	[ ]
1	8	[ ]
2	8	[ ]

`PulseTexture` #1891

Layer	PatternID	Content
0	29	[ ]
1	29	[ ]
2	29	[ ]

ii. Onbeats

This second application will demonstrate how to identify concerted textures where the event type is *`ATTACK`* for all layers in position 0 and not *`ATTACK`* elsewhere. Candidates will be identified using the attacks-profile lookup map, for profiles containing the number of simultaneous events per pulse in pulse position 0 and 0 in all other positions. The number of simultaneous events per pulse is determined by the knowledge-base depth, which is 3.

The patterns will be articulated in three ways:

- Staccato onbeats will be identified using a coverage profile that is the same as the attacks profile.
- Sustained onbeats will be identified using a coverage profile with 3 in all positions.
- Detached onbeats will be identified using a coverage profile with 0 in the rightmost position and with 3 in all other positions.

The query phases of these operations produced 5 candidates of `length` 2, 10 candidates of `length` 3, and 20 candidates of `length` 4.

Filtering by the staccato coverage profile produced exactly three examples of staccato-onbeat textures, one for each texture `length` in the knowledge base:

`PulseTexture` #64

Layer	PatternID	Content
0	2	[▶]
1	2	[▶]
2	2	[▶]

`PulseTexture` #1075

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	13	[▶]

`PulseTexture` #17907

Layer	PatternID	Content
0	29	[▶]
1	29	[▶]
2	29	[▶]

Filtering by the sustained coverage profile produced exactly three examples of sustained-onbeat textures, one for each texture length in the knowledge base:

`PulseTexture` #100

Layer	PatternID	Content
0	4	[▶–]
1	4	[▶–]
2	4	[▶–]

`PulseTexture` #1726

Layer	PatternID	Content
0	20	[▶ ––]
1	20	[▶ ––]
2	20	[▶ ––]

`PulseTexture` #29127

Layer	PatternID	Content
0	62	[▶ –––]
1	62	[▶ –––]
2	62	[▶ –––]

Filtering by the detached coverage profile produced exactly three examples of detached-onbeat textures, one for each texture length in the knowledge base:

`PulseTexture` #64

Layer	PatternID	Content
0	2	[▶–]
1	2	[▶–]
2	2	[▶–]

`PulseTexture` #1605

Layer	PatternID	Content
0	18	[▶ –]
1	18	[▶ –]
2	18	[▶ –]

`PulseTexture` #28551

Layer	PatternID	Content
0	60	[▶ ––]
1	60	[▶ ––]
2	60	[▶ ––]

IX. Application: Additive Rhythm

What I'm looking to do first with this knowledge base is additive rhythm. The most basic thing that happens with additive rhythm is elongating musical ideas one pulse at a time.

This next application will generate a `TextureStatement` compounding six simple texture-statements with this structure:

i. XXO XX'O

Texture-statement **X**, being first, cannot contain *TIE* events in pulse position 0. Its activity profile will be [3,1,2,1]. This profile satisfies the no-starting *TIE* constraint and also establishes a $2+2$ beat structure. Texture-statement **O** will be the staccato onbeat texture of `length` 4, identified previously under "Onbeats" (Subsection ii of Section VIII) as `PulseTexture` #17907. Item **X'** will extend texture-statement **X** by one pulse, added to the end. The first four pulses will have the same content as texture-statement **X**. The fifth pulse will have an *ATTACK* in one layer. This establishes a beat structure of $2+3$.

But wait: The knowledge base does not support `PulseTexture` instances of `length` 5 or greater. That means dividing texture-statement **X** into texture-statement **A** with activity profile [3,1] and texture-statement **B** with activity profile [2,1], then extending texture-statement B into texture-statement B′ with activity profile [2,1,1]. The structure now becomes:

ii. XXO XAB'O

Step 1: Texture-statement **X** is selected by querying the `PulseTexture` supply by attacks profile [3,1,2,1]. This query discovered 100 candidates. Here are the first 6:

`PulseTexture` #17996

Layer	PatternID	Content
0	42	[▶]
1	44	[▶ ▶]
2	50	[▶▶▶▶]

`PulseTexture` #18073

Layer	PatternID	Content
0	42	[▶]
1	46	[▶ ▶–]
2	50	[▶▶▶▶]

`PulseTexture` #18034

Layer	PatternID	Content
0	42	[▶]
1	45	[▶ ▶▶]
2	49	[▶▶▶]

`PulseTexture` #18189

Layer	PatternID	Content
0	42	[▶]
1	49	[▶▶▶]
2	58	[▶–▶▶]

`PulseTexture` #18036

Layer	PatternID	Content
0	42	[▶]
1	45	[▶ ▶▶]
2	51	[▶▶▶–]

`PulseTexture` #18222

Layer	PatternID	Content
0	42	[▶]
1	50	[▶▶▶▶]
2	57	[▶–▶]

The query fetches results in their enumeration order, which seems to disfavor *TIE* events. Since the only criterion prescribed is the attacks profile, any bias introduced by the enumeration algorithm should be overcome by choosing one candidate at random. So I did that (and ended up tweaking the random seed until the result had a few ties). The selected candidate was #19999:

`PulseTexture` #19999

Layer	PatternID	Content
0	44	[▶ ▶]
1	53	[▶▶–▶]
2	57	[▶–▶]

Step 2: PulseTexture #19999 embraces six different actual passages depending upon which musical part plays which layer. The selected permutation orders layers from most to least active, using the *beauty contest* described above under "Order of Patterns in Textures" (Section V):

`PulseTexture` #19999

Layer	PatternID	Content
0	53	[▶▶–▶]
1	57	[▶–▶]
2	44	[▶ ▶]

Step 3: How to divide **X** into **A** and **B**, given how the knowledge base documents relations between `PulseTexture` instances? There is no direct way to extract two out of four pulses from a texture. However, texture-statement **A** can be obtained from texture-statement **X** by looking up an intermediate texture-statement **Q** bearing the relation *TRUNCATE*(3) to **X**, then looking up a `PulseTexture` bearing the relation *TRUNCATE*(2) to **Q**. Both of these lookups are supported by a `BTreeMap`, making them efficient.

Querying the knowledge base for textures with the *TRUNCATE*(3) relation to `PulseTexture` #19999 fetches back:

`PulseTexture` #1255

Layer	PatternID	Content
0	17	[▶▶–]
1	19	[▶–▶]
2	14	[▶ ▶]

Querying the knowledge base for textures with the *TRUNCATE*(2) relation to `PulseTexture` #1255 gives the result desired for texture-statement **A**:

`PulseTexture` #71

Layer	PatternID	Content
0	17	[▶▶]
1	19	[▶–]
2	14	[▶]

Step 4: Extracting the final two pulses out of texture-statement **X** can be accomplished by looking up an intermediate texture-statement R bearing the relation *TRUNCATE*(0) to **X**, then looking up a `PulseTexture` bearing the relation *TRUNCATE*(0) to **R**.

Querying the knowledge base for textures with the *TRUNCATE*(0) relation to `PulseTexture` #19999 fetches back:

`PulseTexture` #700

Layer	PatternID	Content
0	19	[▶–▶]
1	23	[–▶]
2	10	[▶]

Querying the knowledge base for textures with the *TRUNCATE*(0) relation to `PulseTexture` #700 gives the result desired for texture-statement **B**:

`PulseTexture` #68

Layer	PatternID	Content
0	2	[▶]
1	2	[▶]
2	6	[–▶]

Step 5: Querying the knowledge base for textures with the *EXTEND*(2) relation to texture-statement **B** brought back 18 textures of `length` 3. Filtering these down to attacks profile [2,1,1] produced 6 candidates:

`PulseTexture` #1086

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	24	[– ▶▶]

`PulseTexture` #1086

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	24	[– ▶▶]

`PulseTexture` #1100

Layer	PatternID	Content
0	13	[▶]
1	14	[▶ ▶]
2	23	[– ▶]

`PulseTexture` #1100

Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	23	[– ▶]

`PulseTexture` #1102

Layer	PatternID	Content
0	13	[▶]
1	14	[▶ ▶]
2	25	[– ▶ –]

`PulseTexture` #1102

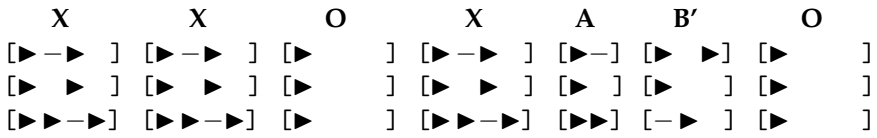
Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	25	[– ▶ –]

Of these `PulseTexture` #1100 was selected at random, then permuted to align with texture-statement **X**:

`PulseTexture` #1100

Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	23	[– ▶]

All the component texture statements have been identified. It just remains to join these simple statements into a compound `TextureStatement` instance. Here is the result:



X. Application: Vertical Masking

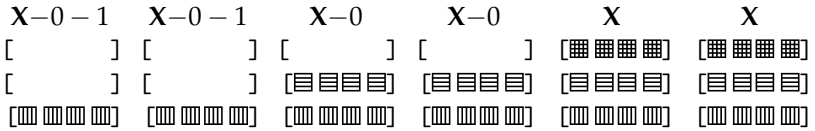
This application gives an imagined nod to the *Schillinger System* [4] or at least something that John Myhill[6] presented as coming from the Schillinger System during a spring 1979 course on composing programs at the State University of New York at Buffalo.

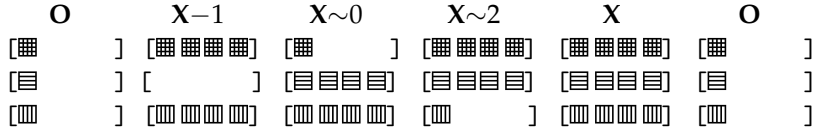
I have mixed feelings about Schillinger. I have no sympathy for his aesthetic, which to me seems to hold that music is somehow 'good' if derivable mathematically and 'poor' otherwise. I am also intimidated both by the sheer size of his tome and by its opaqueness. I acquired the *Schillinger System*'s two massive volumes when still in high school, but never personally got much past the "Theory of Rhythm" (Book I) and the "Theory of Melody" (Book IV). Yet the Schillinger System is no more arbitrary than serialism, and it potentially offers techniques far richer than retrograde, inversion, and transposition. If one can get through those many pages of mathematical ciphers.

The technique of *vertical masking*, employed here, was something John Myhill talked about while presenting Schillinger in the above-mentioned course. The thrust of Myhill's presentation was that one can take a musical passage with several voices and vary it by rotating which instrument plays which part — rotation being Schillinger's go-to permutational device. Vertical masking provides further variation by selectively blanking out some parts while the others continue to play. Since this is exactly what happens in fugal expositions, the processes of part-permutation and vertical masking naturally go together. However I am presently writing this with Schillinger's volumes beside me. Thumbing through these books (which I hadn't cracked in decades), I have not been able to locate the masking scenario Myhill presented.

I had introduced `ROTATE` and `MASK` (the horizontal kind) as `RelationCategory` items with Schillinger in mind. Now I discover that masking is not actually present in Schillinger's technique. I had also been thinking that there was a fair amount of overlap between what I am doing here and what Schillinger did back when. This experience suggests not.

This present application starts with a texture-statement **X**, deconstructs it into layers, then arranges the layers according to the plan presented below:





[6] John Myhill was a close friend of Lejaren Hiller who taught mathematics at SUNY/Buffalo. His field was recursive function theory but he was also keenly interested in composing programs.

O indicates the staccato onbeat texture of `length` 4 identified under "Onbeats" as `PulseTexture` #17907 (Subsection ii of Section VIII). Since the content of **X** has yet to be determined beyond its `length` (4), the above plan uses the symbols "▦", "▤" and "▥" to indicate different layers. Variations upon **X** will be derived by blanking out one or two of the three layers. If "blank" means rest in all pulse positions, then are 3 ways of blanking 1 out of 3 layers and also 3 ways of blanking 2 out of 3 layers. However metric considerations suggest that that for two consecutive statements (a leader and a trailer), then a part which is active in the leader but blank in the trailer should be allowed 'resolution' to the trailer downbeat. So the **X**∼0 and **X**∼1 items in the above plan blank out all bit the first pulse position.

The plan indicates layer-specific blanking options by appending the blanked layer ID, prefixed either by a hyphen ("−") or a tilde ("∼"). The hyphen indicates full silence (all pulses resting), while the tilde indicates an isolated down beat of (just one attack, then rests). For example **X**−0 − 1 indicates the `PulseTexture` which carries over from texture-statement **X** in layer 2 but which is fully silent in layers 0 and 1.

Readers will notice that in spite of my backstory, the plan takes no steps to rotate layers. I originally intended to include rotations but decided it would unnecessarily complicate the plan. Layer permutation is basic functionality in `TextureStatement` instances. It is no reach at all to rotate layers if one wishes to do so.

For this present application, texture-statement **X** will be metric, with full coverage (no part rests during any pulse). It should be equally active in all parts; that is, the `imbalanceCount` (Subsection iv of Section IV) should be 0 (this did not prove attainable).

Step 1: Selecting texture-statement **X** began by querying the `PulseTexture` supply by attacks profile [3,1,2,1]. As reported earlier, this query discovered 100 candidates. Filtering out candidates with coverage profiles other than [3,3,3,3] whittled this number down to 5. None of these candidates had `imbalanceCount` scores of 0, which is what I was hoping for, but three candidates had `imbalanceCount` scores of 1:

`PulseTexture` #24844

Layer	PatternID	Content
0	51	[▶▶▶−]
1	59	[▶−▶−]
2	61	[▶−−▶]

`PulseTexture` #25866

Layer	PatternID	Content
0	53	[▶▶−▶]
1	59	[▶−▶−]
2	59	[▶−▶−]

`PulseTexture` #26306

Layer	PatternID	Content
0	54	[▶▶−−]
1	58	[▶−▶▶]
2	59	[▶−▶−]

The application randomly selected `PulseTexture` #26306, which is fortunate because this texture conforms least slavishly to the meter. (Demonstrating once again that while randomness is necessary for *unbiased* selection, positive criteria should override.) The selected permutation of `PulseTexture` #26306 orders its layers from most to least active, using the *beauty contest* described above under "Order of Patterns in Textures" (Section V):

`PulseTexture` #26306

Layer	PatternID	Content
1	58	[▶ − ▶▶]
2	59	[▶ − ▶ −]
0	54	[▶ ▶ − −]

Statement **X**

The earlier "Silence" application (Subsection i of Section VIII) identified `PulseTexture` #1891 as the silent texture of length 4. The supply of `PulseTexture` instances has a lookup map by pattern IDs, so querying this map with patterns #29 (from all layers of `PulseTexture` #1891), #29 again, and #54 (bottom layer of **X**) is very efficient. This query fetched back `PulsePattern` #1916:

`PulseTexture` #1916

Layer	PatternID	Content
1	29	[ ]
2	29	[ ]
0	54	[▶ ▶ − −]

Statement **X**$-0-1$

The remaining statements are identified by similar lookup queries. The earlier "Onbeats" application (Subsection ii of Section VIII) identified `PulseTexture` #17907 as the staccato onbeat texture of `length` 4. All three layers of `PulseTexture` #17907 employ `PulsePattern` #42, therefore #42 is the 'blanked' pattern ID used in for the **X**~ 0 and **X**~ 2 queries:

`PulseTexture` #2971

Layer	PatternID	Content
0	29	[ ]
1	59	[▶ − ▶ −]
2	54	[▶ ▶ − −]

Statement **X**-0

`PulseTexture` #18350

Layer	PatternID	Content
0	42	[▶]
1	59	[▶ − ▶ −]
2	54	[▶ ▶ − −]

Statement **X**~ 1

`PulseTexture` #2970

Layer	PatternID	Content
0	58	[▶ − ▶▶]
1	29	[ ]
2	54	[▶ ▶ − −]

Statement **X**-1

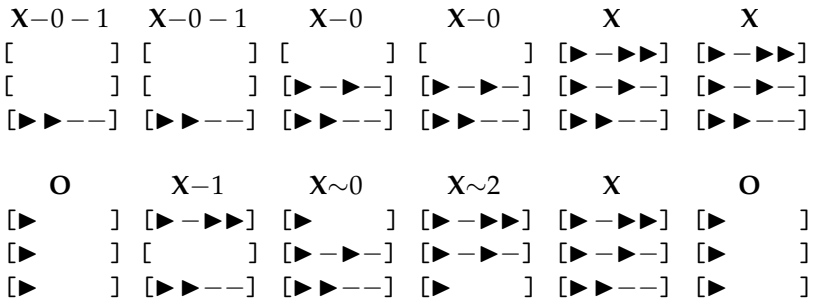
`PulseTexture` #18460

Layer	PatternID	Content
0	58	[▶ − ▶▶]
1	59	[▶ − ▶ −]
2	42	[▶]

Statement **X**~ 2

Understand that the lookup map lists pattern ID's in all possible permutations. Thus looking up the pattern-ID sequence [29,59,54] will fetch back `PulsePattern` #2971 even though this instance actually lists its component patterns in ascending order: [29,54,59]. The code surrounding the lookup request also determines what `Permutation` is necessary to present the patterns in their requested order.

Compounding these seven simple statements together according to the plan graphed above completes the result:



XI. Application: Counter Rhythm

My coined term *counter rhythm* abstracts pitch away from *counter melody*, which according to *Wikipedia* is "a sequence of notes ... written to be played simultaneously with a more prominent lead melody". This next application seeks to write *two* counter rhythms, to be played simultaneously with a more prominent lead rhythm. The lead rhythm is given as an input.

My own take is that a good counter rhythm compliments the lead with respect to the meter. That means that when the lead part syncopates over a strong beat, the counter part fills in the beat. And when the lead part attacks a weak beat, the counter part either ties over or rests. (Remember the premise, stated earlier under "Profiles" (Section VI), that "musical meter is established through the convergence of polyphonic attacks on strong beats and divergence of attacks on weak beats.")

Another desirable feature of counter rhythms is that they actively contribute. This additional proviso was added late after initial attempts produced solutions where one counter rhythm simply rested.

Here is the rhythm for the lead part:

[▶ ▶▶] [− ▶▶ −] [▶]

The meter is described using the attacks profile [2,1,2,1], which repeats. Interpreted with respect to this 2+2 beat structure, the lead rhythm features a syncopation over beat 2. Placing 2, rather than 3, in position 0 of the attacks profile ensures that one counter part will play a pickup rhythm during beat 1. It also ensures that both counter parts will attack the second onbeat (since the lead syncopates there).

The solution involves the following steps:

1. Using the knowledge base to find `PulseTexture` instances which conform to attacks profile [2,1,2,1].

2. Filtering out those `PulseTexture` instances which do not contain the lead rhythm or which have `imbalanceCount` properties greater than 2 (no successful candidates actually had fewer). The results of this step are then randomly shuffled to eliminate enumeration bias.

3. Iterating through the shuffled instances. For each instance, a `Permutation` is sought which brings the lead pattern to part 0 and which also excludes unanticipated ties. If such a permutation is discovered, the `Pulse` and its associated `Permutation` are selected. Otherwise iteration proceeds.

Step 1: Querying for textures with attacks profile [2,1,2,1] returns 476 `PulseTexture` instances.
Step 2: (Filtering)

- Looking up [▶ ▶ ▶] returned `PulsePattern` #45. 58 of the textures returned in Step 1 included `PulsePattern` #45 as one layer while also possessing an `imbalanceCount` of 2.
- Looking up [– ▶ ▶ –] returned `PulsePattern` #72. 20 of the textures returned in Step 1 included `PulsePattern` 4 #72 as one layer while also possessing an `imbalanceCount` of 2.

Step 3: (Iteration and 2nd Filtering)
Here are the first 6 of the 58 shuffled `PulseTexture` instances obtained for [▶ ▶▶] (`PulsePattern` #45).

`PulseTexture` #20853

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	55	[▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #11962

Layer	PatternID	Content
0	36	[▶▶]
1	45	[▶ ▶▶]
2	60	[▶ – –]

`PulseTexture` #21028

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	62	[▶ – – –]
2	72	[– ▶▶ –]

`PulseTexture` #20962

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	59	[▶ – ▶ –]
2	75	[– ▶ – –]

`PulseTexture` #20601

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	47	[▶▶]
2	80	[– – ▶ –]

`PulseTexture` #9507

Layer	PatternID	Content
0	34	[▶]
1	44	[▶ ▶]
2	45	[▶ ▶▶]

The first 3 of these options begin with ties; therefore the specific texture chosen was #11962. This texture lists pattern #45 as layer 1:

`PulseTexture` #11962

Layer	PatternID	Content
1	45	[▶ ▶▶]
0	36	[▶▶]
2	60	[▶ – –]

Here are the first 6 of the 20 shuffled `PulseTexture` instances obtained for [– ▶▶ –] (`PulsePattern` #72).

`PulseTexture` #18956

Layer	PatternID	Content
0	43	[▶ ▶]
1	46	[▶ ▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #27610

Layer	PatternID	Content
0	57	[▶ – ▶]
1	61	[▶ – – ▶]
2	72	[– ▶▶ –]

`PulseTexture` #19308

Layer	PatternID	Content
0	43	[▶ ▶]
1	57	[▶ – ▶]
2	72	[– ▶▶ –]

`PulseTexture` #26754

Layer	PatternID	Content
0	55	[▶ –]
1	58	[▶ – ▶▶]
2	72	[– ▶▶ –]

`PulseTexture` #27185

Layer	PatternID	Content
0	56	[▶ – ▶]
1	69	[▶ – ▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #20853

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	55	[▶ –]
2	72	[– ▶▶ –]

The first of these options is `PulseTexture` #18956. Layer 2 is pattern #72, which is the lead rhythm. This pattern [– ▶▶ –] begins with a *`TIE`*, but that's okay because it follows on after pattern #45 [▶ ▶▶] which does not end with a *`REST`*. So #18956 is the selection:

`PulseTexture` #18956

Layer	PatternID	Content
1	72	[– ▶▶ –]
0	43	[▶ ▶]
2	46	[▶ ▶ –]

And here is the assembled TextureStatement instance:

```
[▶  ▶▶] [– ▶▶ –] [▶    ]
[  ▶▶  ] [▶   ▶] [▶    ]
[▶ – –  ] [▶  ▶ –] [▶    ]
```

XII. Application: Cross Rhythm

The term *cross rhythm* here refers to simultaneous musical parts playing patterns which share a common pulse but which do not share the same length. This is distinguished from *polyrhythm*, where pulses happen at different speeds, and also from *hemiola*, which is a full metric modulation between, say, 3/4 time and 6/8 time. The knowledge base described here copes with hemiola very easily. It does not cope with polyrhythm at all. The present exercise will demonstrate that cross-rhythm is doable. However whether it is worth the trouble depends upon whether one finds it needful to express a cross-rhythm in the `TextureStatement` format.

Cross rhythm and polyrhythm (but not hemiola) are both examples of what Steve Reich calls "phase music". Another example is the selection principle I call "statistical feedback"[7] when applied to nonuniform weights. The compositional attraction here is that you have a period of time over which individual parts proceed inexorably but which the tension between parts destabilizes until everything comes together at the moment of convergence. Like a cadence only different.

This exercise will cross the pattern [▶▶▶▶] (5 pulses) in one musical part with the pattern [▶ – ▶▶] (4 pulses) in a second musical part. Understand that these components need not

[7] https://charlesames.net/feedback/index.html

be literal — for example, one could dynamically swap out [▶▶▶▶] for [▶▶ – ▶] or even [– ▶▶▶▶].

Often just one part 'crosses' the rest of the ensemble; that is, the remaining parts hold to an established beat. The present exercise will not play favorites in that way. Rather the third part will present the rhythm derived by Joseph *Schillinger* in "Interferences of Periodicities", Chapter 2 of Book 1, the Schillinger System's "Theory of Rhythm". In this case the 'periodicities' are 5 and 4 and the resultant sequence of durations (pulse counts) is {4, 1, 3, 2, 2, 3, 1, 4}. The present exercise will articulate these durations in a detached manner, meaning that pulse positions between *ATTACK* events will be *TIE* events up to the position just before the next *ATTACK*; this will be a *REST* event.

What will be produced here is a succession of texture-statement instances, and the first thing to decide is the sequence of statement-instance lengths. One alternative would be employ Schillinger's interference durations; however, the knowledge base does not support `PulseTexture` instances of `length` 1. A reasonable alternative is to employ the shorter pattern `length` (4), since `PulseTexture` instances of `length` 5 are also not supported.

Next follows a collation algorithm which works out what sequence of pulse events each part will play during a statement, which looks up the corresponding `PulsePattern` in the knowledge base, and which in turn uses all three patterns to look up a `PulseTexture` and an associated `Permutation`. I explained how `PulseTexture` lookups work under the "Vertical Masking" application (Section X). The point here is that the knowledge base doesn't really help out with these collation tasks. Cross rhythm is not what the knowledge base is about. The present exercise only makes sense if performed within a larger context that makes active use of the `TextureStatement` representation.

Here is the assembled result:

```
[  ▶▶▶] [▶  ▶▶] [▶▶  ▶] [▶▶▶  ] [▶▶▶▶] [▶     ]
 5          5         5         5
[▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶     ]
 4         4         4         4         4
[▶––  ] [▶▶–  ] [▶  ▶  ] [▶–  ▶] [▶––  ] [▶     ]
 4         1 3       2   2       3   1    4
```

XIII. Reflections

This article describes a personal effort to carry through ideas I've had percolating in response to earlier projects of mine. Whether anybody (other than Schillinger) has done this before, all or in part, I wouldn't know. I have long been out of academics. Having limited cognitively productive hours, I feel no obligation to divert them into scholarship.

I am grateful to Hugo Carvalho, to the MusMat Research Group, and to the *Brazilian Journal of Music and Mathematics* for inviting me to contribute. The software project described here did not exist prior to that invitation, so it can legitimately said to have been created under MusMat auspices.

If I suddenly had a massively parallel supercomputer at my disposal (plus a full-time professional programmer to adapt my existing inline code to multi-threading), I would increase the number of overlaid patterns in a texture (its `depth`) from 3 to 4. That would raise the number of textures by a power of 4 and the number of texture-pair evaluations by a power of 8 (?). I believe

the examples provided by this article demonstrate how the length `limit` can be worked around using compound texture statements. There is no corresponding workaround for the `depth` limit.

Limited changes of scale are imaginable. *Parts* can easily scale vertically to homophonic *choirs*. Creative interpretation of the `PulseEventType` set (`REST`, `ATTACK`, `TIE`) can be used to scale *pulses* horizontally into longer durations. Here context may exert influence. Thus the first duration in the succession [`REST`, `ATTACK`] might be filled with some sort of pickup rhythm.

Something "Concurrence" had which presently does not exist here are primitive pattern alterations like these:

- Promoting a rest/demoting a tie: [►] ⟺ [►–]
- Promoting a tie/demoting an attack: [►–] ⟺ [►►]
- Anticipating/delaying an attack: [►►] ⟺ [►–►]

Such alterations don't readily generalize to textures. Still, they ought to be explored.

References

[1] Ames, Charles (1988). Concurrence. *Journal of New Music Research*, v. 17, n. 1, pp. 3–24.

[2] Ames, Charles (1992). Quantifying Musical Merit. *Journal of New Music Research*, v. 21, n. 1, pp. 53–94.

[3] Kohn, Karl (1981). The Renotation of Polyphonic Music. *The Musical Quarterly*, v. 67, n. 1, pp. 29–49.

[4] Schillinger, Joseph (1941, 1942, 1946). *The Schillinger System of Musical Composition*. New York: Carl Fischer.